5. Reasoning with Horn Clauses

Horn clauses

Clauses are used two ways:
- as disjunctions: \((\text{rain} \lor \text{sleet})\)
- as implications: \((\neg \text{child} \lor \neg \text{male} \lor \text{boy})\)

Here focus on 2nd use

Horn clause = at most one +ve literal in clause
- positive / definite clause = exactly one +ve literal
  e.g. \([\neg p_1, \neg p_2, \ldots, \neg p_n, q]\)
- negative clause = no +ve literals
  e.g. \([\neg p_1, \neg p_2, \ldots, \neg p_n]\) and also \([\,]\)

Note: \([\neg p_1, \neg p_2, \ldots, \neg p_n, q]\) is a representation for
\((\neg p_1 \lor \neg p_2 \lor \ldots \lor \neg p_n \lor q)\) or \([p_1 \land p_2 \land \ldots \land p_n \supset q]\)

so can read as: If \(p_1\) and \(p_2\) and \(\ldots\) and \(p_n\) then \(q\)
and write as: \(p_1 \land p_2 \land \ldots \land p_n \Rightarrow q\) or \(q \Leftarrow p_1 \land p_2 \land \ldots \land p_n\)
Resolution with Horn clauses

Only two possibilities:

It is possible to rearrange derivations of negative clauses so that all new derived clauses are negative

Further restricting resolution

Can also change derivations such that each derived clause is a resolvent of the previous derived one (negative) and some positive clause in the original set of clauses

• Since each derived clause is negative, one parent must be positive (and so from original set) and one parent must be negative.

• Chain backwards from the final negative clause until both parents are from the original set of clauses

• Eliminate all other clauses not on this direct path

This is a recurring pattern in derivations

• See previously:
  – example 1, example 3, arithmetic example

• But not:
  – example 2, the 3 block example
**SLD Resolution**

An SLD-derivation of a clause $c$ from a set of clauses $S$ is a sequence of clause $c_1$, $c_2$, ... $c_n$ such that $c_n = c$, and

1. $c_1 \in S$
2. $c_{i+1}$ is a resolvent of $c_i$ and a clause in $S$

Write: $S \xrightarrow{\text{SLD}} c$

Note: SLD derivation is just a special form of derivation and where we leave out the elements of $S$ (except $c_1$)

In general, cannot restrict ourselves to just using SLD-Resolution

Proof: $S = \{ [p, q], [p, \neg q], [\neg p, q], [\neg p, \neg q] \}$. Then $S \rightarrow \emptyset$.

Need to resolve some $[ \rho ]$ and $[ \bar{\rho} ]$ to get $\emptyset$.

But $S$ does not contain any unit clauses.

So will need to derive both $[ \rho ]$ and $[ \bar{\rho} ]$ and then resolve them together.

**Completeness of SLD**

However, for Horn clauses, we can restrict ourselves to SLD-Resolution

**Theorem:** SLD-Resolution is refutation complete for Horn clauses: $H \rightarrow \emptyset$ iff $H \xrightarrow{\text{SLD}} \emptyset$

So: $H$ is unsatisfiable iff $H \xrightarrow{\text{SLD}} \emptyset$

This will considerably simplify the search for derivations

Note: in Horn version of SLD-Resolution, each clause in the $c_1$, $c_2$, ..., $c_n$, will be negative

So clauses $H$ must contain at least one negative clause, $c_I$
and this will be the only negative clause of $H$ used.

Typical case:
- KB is a collection of positive Horn clauses
- Negation of query is the negative clause
Example 1 (again)

KB

FirstGrade ⊃ Child
Child ∧ Male ⊃ Boy
Kindergarten ⊃ Child
Child ∧ Female ⊃ Girl
Female

Show KB ∪ {¬Girl} unsatisfiable

SLD derivation

alternate representation

A goal tree whose nodes are atoms, whose root is the atom to prove, and whose leaves are in the KB

Prolog

Horn clauses form the basis of Prolog

Append(nil,y,y)
Append(x,y,z) ⇒ Append(cons(w,x),y,cons(w,z))

What is the result of appending [c] to the list [a,b]?

With SLD derivation, can always extract answer from proof

H |= ∃x α(x)
iff
for some term t, H |= α(t)

Different answers can be found by finding other derivations

So goal succeeds with u = cons(a,cons(b,cons(c,nil)))
that is: Append([a b],[c],[a b c])
Back-chaining procedure

Solve\([q_1, q_2, \ldots, q_n]\) = /* to establish conjunction of \(q_i\) */
If \(n=0\) then return \textbf{YES}; /* empty clause detected */
For each \(d \in \text{KB}\) do
  If \(d = [q_1, \neg p_1, \neg p_2, \ldots, \neg p_m]\) /* match first \(q\) */
  and
  /* replace \(q\) by -ve lits */
  Solve\([p_1, p_2, \ldots, p_m, q_2, \ldots, q_n]\) /* recursively */
  then return \textbf{YES}
end for; /* can't find a clause to eliminate \(q\) */
Return \textbf{NO}

Depth-first, left-right, back-chaining

- depth-first because attempt \(p_i\) before trying \(q_i\)
- left-right because try \(q_i\) in order, 1, 2, 3, ...
- back-chaining because search from goal \(q\) to facts in KB \(p\)

This is the execution strategy of Prolog

First-order case requires unification \textit{etc.}

---

Problems with back-chaining

Can go into infinite loop

tautologous clause: \([p, \neg p]\) (corresponds to Prolog program with \(p : - p\)).

Previous back-chaining algorithm is inefficient

Example: Consider \(2n\) atoms, \(p_0, \ldots, p_{n-1}\), \(q_0, \ldots, q_{n-1}\) and \(4n-4\) clauses
\((p_i \Rightarrow p_j), (q_i \Rightarrow p_j), (p_i \Rightarrow q_j), (q_i \Rightarrow q_j)\).

With goal \(p_k\) the execution tree is like this

Is this problem inherent in Horn clauses?
Forward-chaining

Simple procedure to determine if Horn KB |= q.

main idea: mark atoms as solved

1. If q is marked as solved, then return YES
2. Is there a \{p_1, \neg p_2, ..., \neg p_n\} ∈ KB such that
   \(p_2, ..., p_n\) are marked as solved, but the
   positive lit \(p_1\) is not marked as solved?
   no: return NO
   yes: mark \(p_1\) as solved, and go to 1.

FirstGrade example:
Marks: FirstGrade, Child, Female, Girl then done!

Observe:
• only letters in KB can be marked, so at most a linear number of iterations
• not goal-directed, so not always desirable
• a similar procedure with better data structures will run in linear time overall

First-order undecidability

Even with just Horn clauses, in the first-order case we still have
the possibility of generating an infinite branch of resolvents.

KB:
\[
\text{LessThan}(\text{succ}(x), y) \Rightarrow \text{LessThan}(x, y)
\]

Query:
\[
\text{LessThan}(\text{zero}, \text{zero})
\]

As with full Resolution, there is no way to detect
when this will happen

There is no procedure that will test for the
satisfiability of first-order Horn clauses
the question is undecidable

As with non-Horn clauses, the best that we can do is to give
control of the deduction to the user

to some extent this is what is done in Prolog,
but we will see more in “Procedural Control”